Вивчить собі Хаскела на велике щастя!

Автор: Міран Ліповача

Переклад здійснили: Ганна Лелів, Семен Тригубенко, Богдан Пеньковський, Марина Стрельчук і Тетяна Богдан

Мовні редактори: Тетяна Богдан і Ганна Лелів
Науковий редактор: Семен Тригубенко

Переклад виконано за підтримки
Словненя

2017-05-21T00:06:50Z
Версія v4.7-54-gda41cf2

Зміст

14 Застібки-блискавки **1**
14.1 На прогулянці 2
14.2 Хлібні крихти по стежці 5
14.2.1 Назад нагору 6
14.2.2 Маніпуляція деревами, що у фокусі 9
14.2.3 Вгору я біжу прямо на вершину, так-так, де чисте і свіже повітря! 10
14.3 Фокусування на списках 11
14.4 Простенька файлова система 12
14.4.1 Застібка для нашої файлової системи 14
14.4.2 Маніпуляції з нашою файловою системою 16
14.5 Обережно — слизько 17

Покажчик **21**

Розділ 14

Застібки-блискавки

Переклад українською Семена Тригубенка

Той факт, що Хаскел є чистофункційною мовою, має купу переваг, і водночас змушує нас розв'язувати задачі не так, як у нечистофункційних мовах. Завдяки прозорості посилань не можна розрізнити два значення, якщо вони означають одне й те саме.

Отже, якщо ми маємо дерево повне п'ятірок (а чому б і ні?) і хочемо поміняти одну з них на шістку, нам треба якимось чином достеменно знати, яку саме п'ятірку в нашому дереві ми хочемо змінити. Нам треба знати, де вона є у нашому дереві. У нечистофункційних мовах ми могли б просто занотувати, де саме в пам'яті та п'ятірка розташована і поміняти те місце. Але в Хаскелі одна п'ятірка нічим не гірша за іншу, і ми не можемо дискримінувати її за місцем проживання у пам'яті комп'ютера. Ми власне нічого не можемо *змінити*; коли ми кажемо, що міняємо дерево, ми маємо на увазі, що отримуємо одне дерево, а повертаємо — інше, не таке як на вході, але схоже.

Можна зробити ось так: запам'ятовувати шлях від кореня дерева до вузла, який ми хочемо змінити. Скажімо, візьміть оце дерево, йдіть ліворуч, далі — праворуч, а потім — знову ліворуч і поміняйте елемент отам. Так, цей підхід працюватиме, але може бути трохи неефективним. Якщо ми захочемо замінити якийсь вузол поблизу того вузла, який ми щойно змінили, нам треба гуляти тим деревом знову від кореня й до потрібного місця!

У цьому підрозділі ми побачимо, як можна взяти структуру даних і зосередитися на якійсь її частині так, щоб можна було легко змінювати значення по вузлах і ефективно цими вузлами пересуватися. Круто!

14.1 На прогулянці

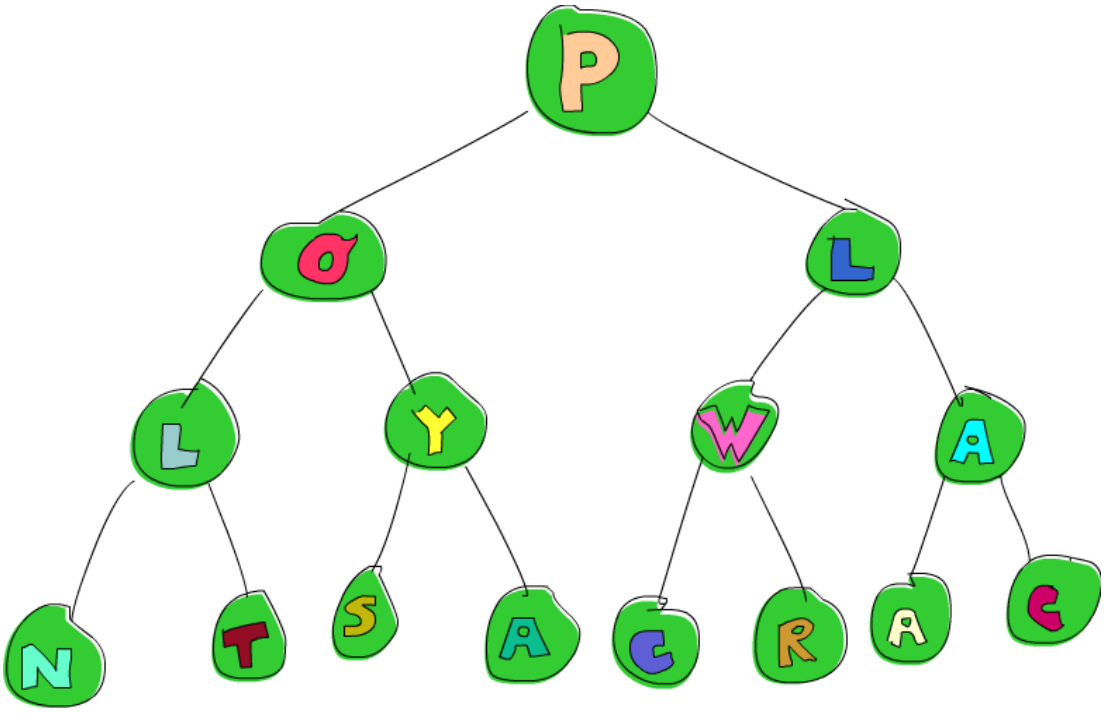
Колись на уроках біології ми вчили, що є багацько різних дерев, тому візьмемо насіння, з якого виросте наше дерево. Ось воно:

```
data Tree a = Empty | Node a (Tree a) (Tree a) deriving (Show)
```

Отже, наше дерево або пусте, або має вузол, що містить елемент-значення і пару піддерев. Ось файний приклад такого дерева, і я даю тобі його, О Мій Читачу, безкоштовно!

```
freeTree :: Tree Char
freeTree =
    Node 'P'
        (Node 'O'
            (Node 'L'
                (Node 'N' Empty Empty)
                (Node 'T' Empty Empty)
            )
            (Node 'Y'
                (Node 'S' Empty Empty)
                (Node 'A' Empty Empty)
            )
        )
        (Node 'L'
            (Node 'W'
                (Node 'C' Empty Empty)
                (Node 'R' Empty Empty)
            )
            (Node 'A'
                (Node 'A' Empty Empty)
                (Node 'C' Empty Empty)
            )
        )
```

А ось тут це дерево представлене графічно:



Зверніть увагу на `W` у тому дереві. Наприклад, якщо ми хочемо поміняти її на `P`. Як то можна зробити? Ну, можна, наприклад, зіставляти взірці по тому дереву, аж доки ми не знайдемо той елемент, що розташовано за адресою: спочатку праворуч, потім ліворуч. Ось потрібний нам код:

```
changeToP :: Tree Char -> Tree Char
changeToP (Node x l (Node y (Node _ m n) r)) = Node x l (Node y (Node 'P' m n) r)
```

Тьху. Виглядає бридко і неохайно, та ще й трохи заплутано. Що тут відбувається? Ми зіставляємо взірець із нашим деревом і називаємо корінь `x` (`x` звертається до `'P'` у корені) і його ліве піддерево — `l`. Замість того, щоб дати ім'я правому піддереву, ми продовжуємо зіставляти по ньому. І продовжуємо в тому дусі аж до того моменту, як дійдемо до піддерева із коренем, що містить наше рідне `'W'`. Як то буде зроблено, ми перебудовуємо наше дерево, але з новим піддеревом, що містить `'W'` у корені замість колишнього `'P'`.

Чи є кращий спосіб? Може створімо функцію, що братиме дерево і путівник по ньому? Путівник міститиме вказівки типу `L` чи `R`, що означатимуть ліворуч і праворуч, відповідно, і ми замінюватимемо елемент у вузлі, до якого потрапляємо, керуючись тими вказівками. Ось воно:

```
data Direction = L | R deriving (Show)
type Directions = [Direction]

changeToP :: Directions-> Tree Char -> Tree Char
changeToP (L:ds) (Node x l r) = Node x (changeToP ds l) r
```

```
changeToP (R:ds) (Node x l r) = Node x l (changeToP ds r)
changeToP [] (Node _ l r) = Node 'P' l r
```

Якщо перший елемент у списку вказівок є `L`, ми будуємо нове дерево точнісінько так, як старе, тільки елемент у його лівому піддереві зміниться на `'P'`. Коли ми рекурсивно викликаємо `changeToP`, ми подаємо як параметри лише хвіст списку вказівок, бо ми вже прислухалися до однієї з них, коли пішли ліворуч. Ми робимо так само у випадку `R`. Якщо список директив пустий, це означає що ми прибули в точку призначення, тому ми повертаємо дерево на кшталт того, що було подано, от лише `'P'` тепер є елементом вузла-кореня того дерева.

Щоб не друкувати усе дерево, створимо функцію, що бере наш список вказівок-директив і каже, який елемент «мешкає» за тією адресою:

```
elemAt :: Directions -> Tree a -> a
elemAt (L:ds) (Node _ l _) = elemAt ds l
elemAt (R:ds) (Node _ _ r) = elemAt ds r
elemAt [] (Node x _ _) = x
```

Ця функція насправді є дуже схожою на `changeToP`, от лише замість запам'ятовування по ходу справи і реконструювання дерева в такий спосіб, вона ігнорує все окрім пункту її призначення. Ось тут ми міняємо `'W'` на `'P'` і бачимо, чи містить те нове дерево наші зміни:

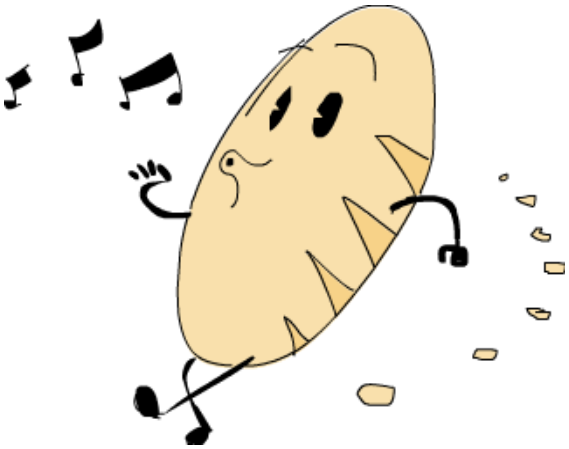
```
ghci> let newTree = changeToP [R,L] freeTree
ghci> elemAt [R,L] newTree
'P'
```

Кльово, здається що спрацювало. У цих функціях список вказівок грає роль *фокусу*, тому що він «фокусує» нашу увагу на потрібному піддереві нашого дерева. Наприклад, список вказівок `[R]` фокусує нашу увагу на піддереві праворуч від кореня. Пустий список вказівок зосереджує нашу увагу на всьому дереві.

Хоча така техніка виглядає модною, вона може бути неефективною, особливо коли ми хочемо міняти елементи один за іншим, багато разів. Хай у нас є величезна тополя і довгий, як жердяка, список вказівок, що веде нас до якогось елемента внизу того дерева. Ми прислуха́ємося до тих вказівок, спускаємося тим деревом і міняємо елемент внизу. Якщо ми хочемо змінити елемент неподалік, маємо починати з коріння і йти знайомим шляхом донизу — увесь шлях, знову. Воловодіння безконечне.

У наступному підрозділі ми знайдемо кращий шлях. Будемо фокусуватися на піддереві, але у спосіб, що даватиме змогу ефективно перемикатися на піддерева неподалік.

14.2 Хлібні крихти по стежці



Добре, — аби сфокусуватися на піддереві, ми хочемо щось ліпше, ніж просто список вказівок, яких ми весь час дотримуємося, починаючи рух від кореня нашого дерева. А чи не буде ліпше, якщо ми почнемо в корені і рухатимемося ліворуч чи праворуч, один крок за одну операцію, і лишатимемо щось на кшталт хлібних крихт? Тобто, коли йдемо ліворуч, пам'ятатимемо, що ми пішли ліворуч, а коли праворуч — то праворуч, відповідно. Звичайно, таке можна спробувати.

Представлення у програмі для такої навігаційної стежки буде схожим на список, що складатиметься з `Direction` (можливі варіанти — `L` чи `R`), от лише не називатимемо його `Directions`. Краще назвемо його `Breadcrumbs`, бо тепер наші вказівки буде подано у зворотньому порядку, оскільки ми залишатимемо їх під час подорожі тим деревом:

```
type Breadcrumbs = [Direction]
```

Ось функція, що бере дерево і стежку і рухається ліворуч, додаючи `L` у голову списку, який представляє стежку:

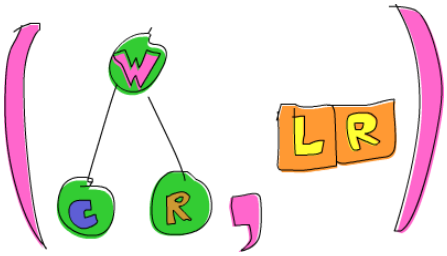
```
goLeft :: (Tree a, Breadcrumbs) -> (Tree a, Breadcrumbs)
goLeft (Node _ l _, bs) = (l, L:bs)
```

Ми ігноруємо елементи в корені і правому піддереві і просто повертаємо ліве піддерево разом із старими крихтами плюс `L` у голові. Ось функція для ходіння праворуч:

```
goRight :: (Tree a, Breadcrumbs) -> (Tree a, Breadcrumbs)
goRight (Node _ _ r, bs) = (r, R:bs)
```

Вона працює так само. Використаймо ці функції і підемо нашим деревом `freeTree` праворуч, а потім ліворуч:

```
ghci> goLeft (goRight (freeTree, []))
(Node 'W' (Node 'C' Empty Empty) (Node 'R' Empty Empty),[L,R])
```



Окей, то тепер ми маємо дерево із `'W'` у корені і `'C'` у корені його лівого піддерева, а `'R'` — у корені правого. Навігаційна стежка є `[L,R]`, оскільки ми пішли спочатку праворуч, а потім — ліворуч.

Щоб зробити прогулянку деревом зрозумілішою, ми можемо задіяти функцію `-:`, що працюватиме ось як:

```
x -: f = f x
```

Це дасть нам змогу викликати функції у новий спосіб, де спочатку ми пишемо аргумент, а потім `-:` і вже після того ім'я функції. Тобто замість `goRight (freeTree, [])`, ми можемо тепер написати `(freeTree, []) -: goRight`. Використовуючи це, ми перепишемо вищенаведене як вираз, із якого легше бачити, що ми йдемо спочатку праворуч, а потім ліворуч:

```
ghci> (freeTree, []) -: goRight -: goLeft
(Node 'W' (Node 'C' Empty Empty) (Node 'R' Empty Empty),[L,R])
```

14.2.1 Назад нагору

А що, якщо ми тепер захочемо піти назад наверх нашим деревом? Із нашої стежки хлібних крихт ми знаємо, що поточне дерево є лівим піддеревом батьківського дерева, а воно, в свою чергу, є правим піддеревом свого, і це все. У нас немає достатньо інформації про батьківське дерево, щоб можна було на нього назад залізти. Здається, що окрім напрямку руху, хлібна крихта має ще й містити решту інформації, потрібної для зворотнього ходу. У цьому випадку, це є елемент з вузла-кореня батьківського дерева разом із його правим піддеревом.

Загалом, одна хлібна крихта має містити усі дані потрібні для відбудови батьківського вузла. Тому там має бути інформація про всі шляхи, якими ми не пішли, і має також бути відомо про напрям, у якому ми пішли. А от інформація про піддерево, на якому ми сфокусувалися, нам не потрібна. Адже ми вже маємо те піддерево як першу компоненту кортежу, тому якби вона також була в хлібній крихті, ми б зберігали цю інформацію двічі.

Модифікуймо наші крихти, щоб вони також містили інформацію про все, чим ми знехтували, коли рухалися ліворуч чи праворуч. Замість `Direction`, ми побудуємо новий тип даних:

```
data Crumb a = LeftCrumb a (Tree a)
```

```
              | RightCrumb a (Tree a) deriving (Show)
```

Тепер замість простого `L` маємо `LeftCrumb`, що містить елемент вузла, з якого ми прийшли, і праве піддерево, яке ми не відвідали. Замість `R` — маємо `RightCrumb`, що містить елемент з вузла, з якого ми прийшли, і ліве піддерево, куди ми не завітали.

Ця навігаційна хрустка смакота тепер містить усе що треба, щоб відбудувати дерево, яким ми спустилися. Тому замість хлібних крихт ми тепер користуємося чимось на кшталт дискет, які ми залишаємо на нашому шляху, і які містять набагато більше інформації порівняно з просто напрямом, який ми обрали.

По суті, кожна крихта тепер є деревом із діркою. Коли ми заглиблюємося у дерево, хлібокрихта зберігає усю інформацію, яку зберігав вузол, із якого ми прийшли, *за винятком* піддерева, яке ми вибрали для фокусування. До того ж, місце з діркою позначено. У випадку `LeftCrumb`, ми знаємо що ми рушили ліворуч, тому піддерево, якого не вистачає, є лівим піддеревом.

Змінимо також наш синонімічний тип `Breadcrumbs`, щоб скористатися нашими нововведеннями:

```
type Breadcrumbs a = [Crumb a]
```

Рухаючись нагору, нам треба оновити функції `goLeft` і `goRight` і зберігати інформацію про шляхи, якими ми не пішли, а не просто знехтувати ними, як ми робили раніше. Ось, `goLeft`:

```
goLeft :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goLeft (Node x l r, bs) = (l, LeftCrumb x r:bs)
```

Видно, що вона дуже схожа на її попередника `goLeft`, от лише замість простого додавання `L` у голову навігаційної стежки, ми додаємо `LeftCrumb`, щоб зазначити, що ми пішли ліворуч, і навантажуємо нашу `LeftCrumb` елементом вузла, з якого подорожуємо (це є `x`), й правим піддеревом, яке ми вирішили не відвідувати.

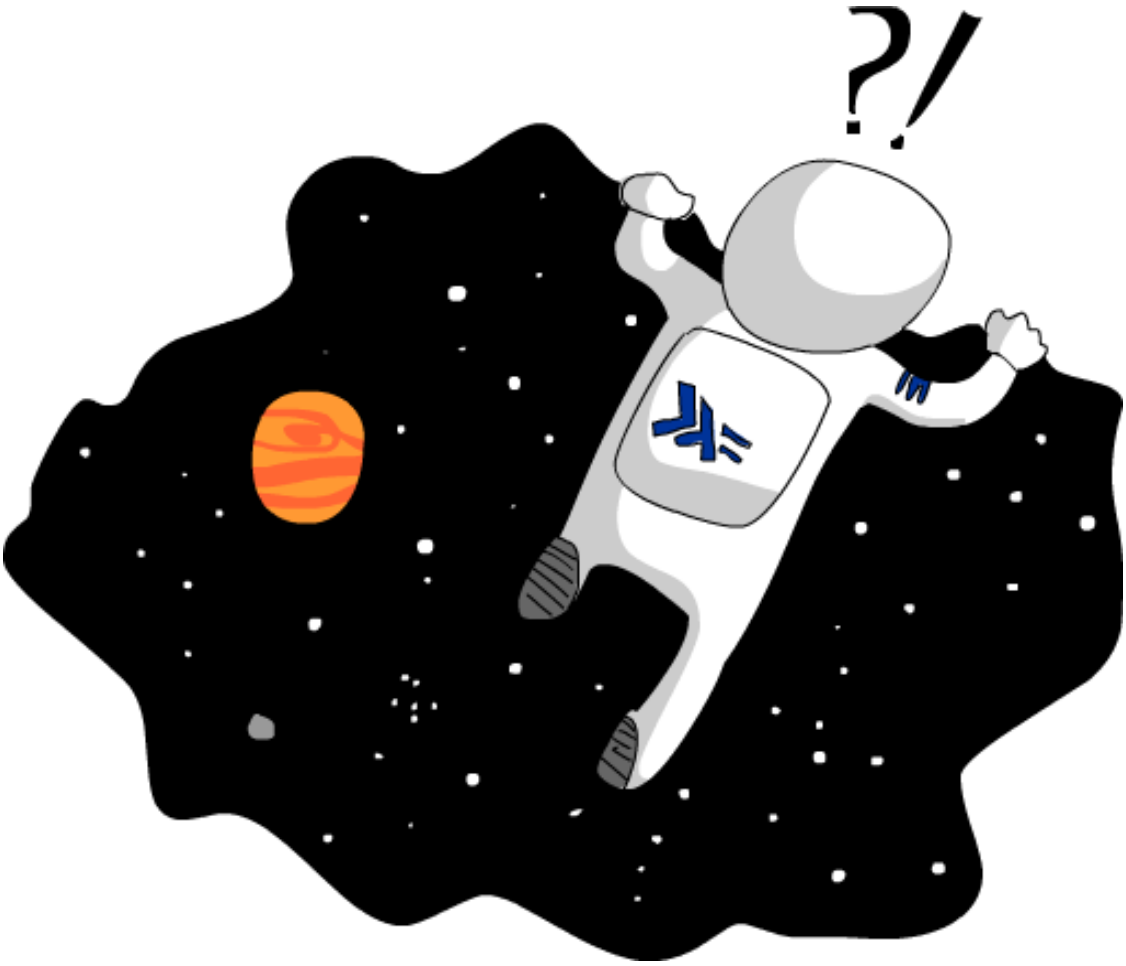
Зверніть увагу, що ця функція спирається на припущення, що поточне дерево в фокусі не є `Empty`. Пусте дерево не має піддерев, тому якщо ми спробуємо піти ліворуч пустим деревом, отримаємо помилку, позаяк зіставлення по `Node` не спрацює, а взірця, який подбає про `Empty`, іще не означено.

`goRight` є подібною:

```
goRight :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goRight (Node x l r, bs) = (r, RightCrumb x l:bs)
```

Раніше ми мали змогу йти ліворуч або праворуч. А тепер ми маємо змогу йти назад наверх, бо ми запам'ятали різні речі про батьківські вузли і шляхи, що їх не відвідали. Ось код функції `goUp`:

```
goUp :: (Tree a, Breadcrumbs a) -> (Tree a, Breadcrumbs a)
goUp (t, LeftCrumb x r:bs) = (Node x t r, bs)
goUp (t, RightCrumb x l:bs) = (Node x l t, bs)
```



Ми фокусуємося на дереві `t` і перевіряємо найновішу `Crumb`. Якщо вона є `LeftCrumb`, тоді ми будуємо нове дерево із нашим деревом `t` у якості лівого піддерева, і використовуємо інформацію про праве піддерево, що ми не відвідали, і елемент для добудови `Node`. Оскільки ми повернулися назад і, так би мовити, підняли останню крихту і використали її для відбудови батьківського дерева, новий список хлібних крихт її не міститиме.

Зверніть увагу, що ця функція завалюється, якщо ми нагорі дерева і спробуємо залізти ще вище. Незабаром ми скористаємося монадою `Maybe`, щоб описати можливі аварії зміни фокуса.

`Tree a` і `Breadcrumbs a` — то є вся потрібна інформація для відбудови усьо-

го дерева. Крім того, ми запам'ятовуємо, на якому піддереві сфокусувалися. Така схема уможливлює легкі рухи вгору, ліворуч чи праворуч. Така парочка плюс фокус на частину структури даних плюс її оточення називається застібкою-блискавкою, бо рух фокусу догори і вниз по структурі даних нагадує роботу звичайної застібки на звичайних штанях. Тому тип-синонім файно назвати ось як:

```
type Zipper a = (Tree a, Breadcrumbs a)
```

Я волів би назвати той тип-синонім `Focus`, бо така назва підкреслює, що ми фокусуємося на частині структури даних. Однак термін застібка є більш розповсюдженим, то ми не вигадуватимемо колесо і використовуватимемо `Zipper`.

14.2.2 Маніпуляція деревами, що у фокусі

Тепер ми вже можемо рухатися вниз і вгору, тому напишімо функцію, яка модифікує елемент у корінні піддерева, на яке вказує застібка:

```
modify :: (a -> a) -> Zipper a -> Zipper a
modify f (Node x l r, bs) = (Node (f x) l r, bs)
modify f (Empty, bs) = (Empty, bs)
```

Якщо ми фокусуємося на вузлі, ми модифікуємо його корінь функцією `f`. Якщо ми фокусуємося на пустому піддереві, ми лишаємо його як є. Тепер ми можемо розпочати роботу з деревом, перейти по ньому куди треба і модифікувати елемент, тримаючи фокус на тому елементі, щоб ми могли легко перейти вниз чи догори, за потребою. До прикладу:

```
ghci> let newFocus = modify (\_ -> 'P') (goRight (goLeft (freeTree,[])))
```

Ми йдемо ліворуч, потім праворуч і модифікуємо елемент в корені, замінюючи його на `'P'`. Читається краще, якщо задіяти `-:`:

```
ghci> let newFocus = (freeTree,[]) -: goLeft -: goRight -: modify (\_ -> 'P')
```

Ми можемо перейти вверх, якщо забажаємо, і замінити елемент на таємниче `'X'`:

```
ghci> let newFocus2 = modify (\_ -> 'X') (goUp newFocus)
```

Або ж, якщо написати це з `-:`:

```
ghci> let newFocus2 = newFocus -: goUp -: modify (\_ -> 'X')
```

Рухатися нагору так легко завдяки хлібним крихтам, що описують структуру даних, на якій ми не фокусуємося. Тільки тепер її вивернуто, як таку собі

шкарпетку. Ось тому, якщо ми хочемо піднятися нагору, нам не треба починати з кореня і спускатися, бо ми просто знімаємо верхівку нашого інвертованого дерева, і таким чином де-інвертуємо якусь його частину і додаємо її до нашого фокуса.

Кожен вузол має два піддерева, навіть якщо ті піддерева пусті. Отже, якщо ми фокусуємося на пустому піддереві, ми можемо замінити його на непусте, і, таким чином, під'єднаємо дерево до вузла-листя. Код, що реалізує це, — простий:

```
attach :: Tree a -> Zipper a -> Zipper a
attach t (_, bs) = (t, bs)
```

Ми беремо дерево і застібку та повертаємо нову застібку, фокус якої замінено на подане дерево. Ми не тільки можемо розширяти дерева у такий спосіб, себто, заміняючи піддерева новими деревами, а й можемо заміняти цілі, вже існуючі, піддерева. Під'єднаймо дерево у лівий нижній кут нашого `freeTree`:

```
ghci> let farLeft = (freeTree,[]) -: goLeft -: goLeft -: goLeft -: goLeft
ghci> let newFocus = farLeft -: attach (Node 'Z' Empty Empty)
```

`newFocus` тепер вказує на дерево, яке ми щойно причепили, а решта піддерев лежать у хлібних крихтах. Якщо б ми зійшли нагору за допомогою `goUp`, то мали б те саме дерево `freeTree`, але з додатковим `'Z'` внизу, з лівої сторони.

14.2.3 Вгору я біжу прямо на вершину, так-так, де чисте і свіже повітря!

Написати функцію, що іде наверх деревом, незалежно від поточного фокусу, дуже легко. Ось вона:

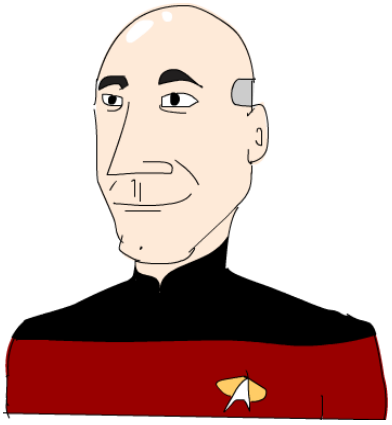
```
topMost :: Zipper a -> Zipper a
topMost (t,[]) = (t,[])
topMost z = topMost (goUp z)
```

Якщо наша навігаційна стежка на стероїдах пуста, це означає, що ми опинилися на вершині нашого дерева, тому ми просто повертаємо поточний фокус. В іншому випадку, ми йдемо нагору, щоб отримати фокус на батьківський вузол і рекурсивно викликаємо `topMost` по ньому. Отож тепер ми можемо ходити по нашому дереву як нам заманеться, ліворуч чи праворуч чи нагору, застосовуючи `modify` і `attach` коли треба, а по завершенню тих модифікацій, ми пересуваємо фокус на коріння, використовуючи функцію `topMost` і бачимо наші зміни в звичній нам «проекції».

14.3 Фокусування на списках

Застібки можна використовувати з будь-якою структурою даних, тому не дивно, що вони застосовуються для фокусування на підсписках списків. Врешті-решт, списки дуже схожі на дерева — у вузлі дерева є елемент (чи немає) і декілька піддерев, а «вузол» списку має елемент і лише один підсписок. Коли ми реалізували наші власні списки в підрозділі ??, ми означили цей тип даних ось як:

```
data List a = Empty | Cons a (List a) deriving (Show, Read, Eq, Ord)
```



Порівняйте це з нашим означенням бінарного дерева — відразу видно, що можна думати про списки, як про дерева, де вузли мають лише одне піддерево.

А списки на кшталт `[1,2,3]` можна написати як `1:2:3:[]`. Такі списки складаються з голови списку, тобто, в цьому випадку, `1` і власне хвоста того списку, що дорівнює `2:3:[]`. `2:3:[]` теж має голову, тобто `2` і хвіст `3:[]`. У `3:[]` голова є `3`, а хвіст — пустий список `[]`.

Створімо застібку для списків. Щоб змінити фокус у списках, ми рухаємося або вперед, або назад (а для дерев можливі рухи були: вниз ліворуч, вниз праворуч або нагору). Частина у фокусі буде підсписком, і разом із нею ми лишатимемо хлібні крихти у міру того, як просуватимемося уперед. Ну а з чого складатиметься одна спискова крихта? Коли ми працювали з бінарними деревами, то казали, що хлібна крихта має містити елемент батьківського вузла, разом із усіма піддеревами того вузла, які ми не вибрали для фокусування. Також крихта зазначала, чи ми пішли ліворуч чи праворуч. Отже, там було все окрім піддерева у фокусі.

Списки простіші за дерева, тому нам не треба пам'ятати, ліворуч ми пішли чи ні, оскільки є лише один шлях заглибитися у список. Оскільки кожен вузол має лише один-єдиний підсписок, нам також не треба пам'ятати шляхи, що їх ми не обрали. Виходить, що нам треба тільки запам'ятати попередній елемент. Якщо ми маємо список на кшталт `[3,4,5]` і знаємо, що попередній елемент був `2`, ми можемо піти назад, просто повернувши той елемент до голови нашого списку, отримуючи `[2,3,4,5]`.

Оскільки хлібна крихта містить лише значення, нам не обов'язково запаковувати її всередину типу даних, як ми робили, коли створювали тип даних `Crumb` для застібок для дерев:

```
type ListZipper a = ([a],[a])
```

Перший список то є підсписок у фокусі, а другий — список хлібних крихт. Напишімо функцію, що ходить вперед і назад по списках:

```
goForward :: ListZipper a -> ListZipper a
goForward (x:xs, bs) = (xs, x:bs)

goBack :: ListZipper a -> ListZipper a
goBack (xs, b:bs) = (b:xs, bs)
```

Коли ми йдемо вперед, то фокусуємося на хвості поточного списку і залишаємо значення з голови як крихту. Коли вертаємося, ми беремо найсвіжішу крихту і вставляємо її у голову списку.

Ну а тепер поглянемо на ці функції у дії:

```
ghci> let xs = [1,2,3,4]
ghci> goForward (xs,[])
([2,3,4],[1])
ghci> goForward ([2,3,4],[1])
([3,4],[2,1])
ghci> goForward ([3,4],[2,1])
([4],[3,2,1])
ghci> goBack ([4],[3,2,1])
([3,4],[2,1])
```

Бачимо, що навігаційна стежка у випадку списків то є просто розвернута попередня частина нашого списку. Значення, від якого ми віддаляємося, завжди йде до голови списку крихт, тому легко повернутися, перекинувши його назад з голови списку крихт до голови списку у фокусі.

Тут добре видно, чому ці штуки називають застібками-блискавками: бо вони нагадують такі застібки повзунком, що рухається вверх-вниз.

Якби ми з вами писали текстовий редактор, можна було б узяти список рядків як представлення рядків тексту. Тоді за допомогою блискавки ми би позначили лінію, на якій стоїть курсор. Використання блискавки також полегшило б додавання нових рядків тексту будь-де у тексті, або ж видалення вже існуючих рядків тексту.

14.4 Простенька файлова система

Тепер, коли ми вже знаємо, як працюють блискавки, використаймо дерева, щоб описати файлову систему, а тоді створімо блискавку для тієї файлової системи,

щоб можна було пересуватися з директорії в директорію, точнісінько так, як ми зазвичай це робимо, стрибаючи по власній файловій системі.

Кажучи по-простому, файлова система складається переважно з файлів і директорій. Файли — це одиниці даних, вони мають ім'я, тоді як директорії використовуються для організації тих файлів і можуть містити файли або ж інші директорії. Тобто одиниця файлової системи є або файлом, що має ім'я і якісь дані, або директорією, що має ім'я і зміст — купу інших одиниць, — файлів чи директорії. Ось тип даних для реалізації цього і деякі типи-синоніми, аби ми знали, що є що:

```
type Name = String
type Data = String
data FSItem = File Name Data | Folder Name [FSItem] deriving (Show)
```

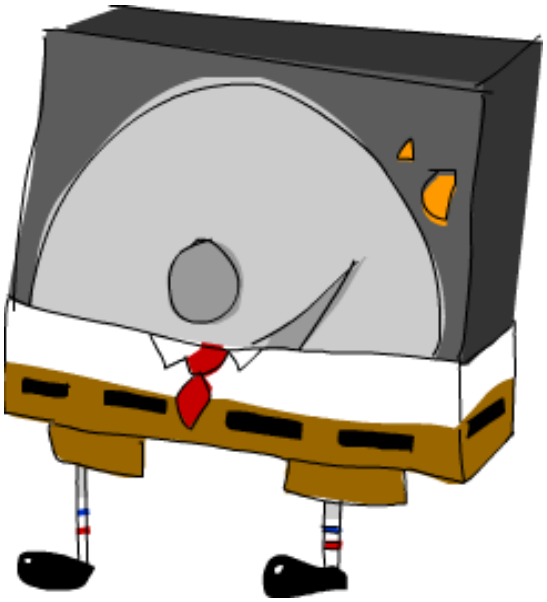
Файл складається з двох рядків — його імені і даних, які він тримає у собі. Директорія складається з рядка з ім'ям і списку-змісту. Якщо той список пустий, маємо пусту директорію.

Ось директорія із декількома файлами і піддиректоріями:

```
myDisk :: FSItem
myDisk =
    Folder "root"
        [ File "goat_yelling_like_man.wmv" "baaaaaa"
        , File "pope_time.avi" "god bless"
        , Folder "pics"
            [ File "ape_throwing_up.jpg" "bleargh"
            , File "watermelon_smash.gif" "smash!!"
            , File "skull_man(scary).bmp" "Yikes!"
            ]
        , File "dijon_poupon.doc" "best mustard"
        , Folder "programs"
            [ File "fartwizard.exe" "10gotofart"
            , File "owl_bandit.dmg" "mov eax, h00t"
            , File "not_a_virus.exe" "really not a virus"
            , Folder "source code"
                [ File "best_hs_prog.hs" "main = print (fix error)"
                , File "random.hs" "main = print 4"
                ]
            ]
        ]
```

Так виглядає зміст мого диску на теперішній момент.

14.4.1 Застібка для нашої файлової системи



Тепер у нас є файлова система, і все що нам залишилося — це застібка для легкої навігації по ній, щоб, своєю чергою, ми могли модифікувати і видаляти файли та директорії. Як із бінарними деревами і списками, ми залишатимемо хлібні крихти з інформацією про всі місця, які ми вирішили не відвідувати. Тобто одна така крихта повинна бути чимось на кшталт вузла, от лише містити вона має все, окрім піддерева у фокусі. Також вона має містити інформацію про позицію дірки, щоб ми знали куди вставляти наш попередній фокус, як рухатимемося нагору.

У цьому випадку крихта має бути звичайною директорією, от тільки вона не повинна містити директорію, яку ми вибрали. Ви запитаєте: а чому крихта є директорією, а не файлом? Ну, тому що, коли ми сфокусувалися на файлі, ми вже не можемо спускатися глибше по файловій системі, тому немає сенсу лишати крихту, яка каже, що ми прийшли з файлу. Файл то є щось на кшталт пустого піддерева.

Якщо ми фокусуємося на директорії `"root"`, а потім на файлі `"dijon_poupon.doc"`, як виглядатиме крихта, що її ми залишаємо? Вона повинна містити ім'я батьківської директорії та її зміст, що складатиметься з двох частин — змісту, що передує файлові, на якому ми сфокусувалися, і решту змісту, що йде після нього. Отже, все що нам треба — це `Name` і два списки для змісту. Тримаючи зміст у двох списках, ми знаємо, де саме є те місце, куди ми спустилися, тому коли знову рухатимемося нагору, знатимемо, де покласти те, чого бракує. Тобто ми знаємо, де сидить дірка.

Ось тип крихти для файлової системи:

```
data FSCrumb = FSCrumb Name [FSItem] [FSItem] deriving (Show)
```

А ось тип-синонім для нашої застібки:

```
type FSZipper = (FSItem, [FSCrumb])
```

Іти назад нагору по цій ієрархії дуже просто. Ми просто беремо найсвіжішу крихту і збираємо новий фокус з поточного фокусу і неї. Ось так:

```
fsUp :: FSZipper -> FSZipper
fsUp (item, FSCrumb name ls rs:bs) = (Folder name (ls ++ [item] ++ rs), bs)
```

Наша крихта знає ім'я батьківської директорії, елементи, які передували елементові у фокусі (тобто `ls`) і елементи після нього (тобто `rs`), тому нам легко рухатися нагору.

Може заглибимося у файлову систему ще трохи? Якщо ми в `"root"` і хочемо сфокусуватися на `"dijon_poupon.doc"`, хлібні крихти, які ми залишаємо, будуть містити ім'я `"root"` разом із усіма елементами, що передують `"dijon_poupon.doc"`, і усіма, що йдуть після нього.

Ось функція, яка бере ім'я і фокусується на файлі чи директорії, що знаходиться у директорії у поточному фокусі:

```
import Data.List (break)

fsTo :: Name -> FSZipper -> FSZipper
fsTo name (Folder folderName items, bs) =
    let (ls, item:rs) = break (nameIs name) items
    in  (item, FSCrumb folderName ls rs:bs)

nameIs :: Name -> FSItem -> Bool
nameIs name (Folder folderName _) = name == folderName
nameIs name (File fileName _) = name == fileName
```

`fsTo` бере `Name` і `FSZipper` і повертає нову `FSZipper`, що вказує на файл із відповідним ім'ям. Цей файл має бути у директорії у поточному фокусі. Ця функція не шукатиме той файл скрізь, вона лише перегляне зміст поточної директорії.

Спершу ми використаємо `break`, щоб розділити список змісту директорії на дві частини — ту, що передує файлу, який ми шукаємо, і ту частину, що йде після нього. Якщо ви пригадуєте, `break` бере предиката і список і повертає пару списків. Перший список у парі міститиме всі елементи, для яких предиката повернула `False`. Щойно предиката повернула `True` для якогось елемента, функція повертає той елемент і решту того списку як список номер два у парі. Ми написали допоміжну функцію під назвою `nameIs`, що бере ім'я і одиницю файлової системи і повертає `True`, якщо імена співпадають.

Тепер `ls` — це список, що містить елементи, які передують тому, що ми шукаємо, `item` — це те, що ми шукаємо, і `rs` — це список з елементів, які йдуть після того, що ми шукаємо. Тепер коли в нас це все є, ми просто видаємо те, що нам повернула функція `break` і будуємо крихту, що має усі дані, яких потребує.

Зверніть увагу: якщо ім'я, яке ми шукаємо, не є присутнім у директорії, взірець `item:rs` спробує зіставитись із пустим списком, і ми отримаємо помилку. Також, якщо наш поточний фокус взагалі не директорія, а файл, ми також матимемо помилку і ще й аварійне завершення програми.

Тепер ми можемо пересуватися вгору і вниз файловою системою. Почнімо з кореня і підемо до файлу `"skull_man(scary).bmp"`:

```
ghci> let newFocus =
          (myDisk,[]) -: fsTo "pics" -: fsTo "skull_man(scary).bmp"
```

`newFocus` тепер є застібкою, що вказує на файл `"skull_man(scary).bmp"`. Перегляньмо першу компоненту застібки (тобто, власне, фокус), щоб переконатися, що це дійсно так:

```
ghci> fst newFocus
File "skull_man(scary).bmp" "Yikes!"
```

Перейдімо нагору і сфокусуймося на сусідньому файлі `"watermelon_smash.gif"`:

```
ghci> let newFocus2 = newFocus -: fsUp -: fsTo "watermelon_smash.gif"
ghci> fst newFocus2
File "watermelon_smash.gif" "smash!!"
```

14.4.2 Маніпуляції з нашою файловою системою

Тепер ми знаємо, як рухатися нашою файловою системою, і тому зможемо легко нею маніпулювати. Ось функція для перейменування файлу чи директорії, яка зараз у фокусі:

```
fsRename :: Name -> FSZipper -> FSZipper
fsRename newName (Folder name items, bs) = (Folder newName items, bs)
fsRename newName (File name dat, bs) = (File newName dat, bs)
```

Тепер ми можемо змінити ім'я директорії `"pics"` на `"cspi"`:

```
ghci> let newFocus =
          (myDisk,[]) -: fsTo "pics" -: fsRename "cspi" -: fsUp
```

Ми спустилися до директорії `"pics"`, змінили її і повернулися назад нагору.

А як щодо функції, яка створює новий елемент у нашій поточній директорії? Дивіться і дивуйтеся:

```
fsNewFile :: FSItem -> FSZipper -> FSZipper
fsNewFile item (Folder folderName items, bs) =
    (Folder folderName (item:items), bs)
```

Просто, як двері. Зверніть увагу, що ця функція завалиться, якщо ми спробуємо додати новий елемент, а сфокусуємося на файлі, а не на директорії.

Додаймо до нашої директорії `"pics"` якийсь файл і після того повернімося у корінь:

```
ghci> let newFocus =
          (myDisk,[]) -: fsTo "pics" -:
              fsNewFile (File "heh.jpg" "lol") -: fsUp
```

Найкрутіша річ з цього усього — це те, що коли ми модифікуємо нашу файлову систему, ми не модифікуємо оригінал, а повертаємо новеньку файлову систему з нашими змінами. У такий спосіб ми маємо доступ як до старої файлової системи (у цьому випадку, `myDisk`) так і до нової (перша компонента `newFocus`). Отже, користуючись застібками, ми отримуємо систему керування версіями як бонус. Іншими словами, ми завжди можемо звернутися до старої версії структур даних після їхньої модифікації, так би мовити. Це не є унікальним для застібок — це властивість Хаскела, бо його структури даних не можна змінити. Завдяки застібкам у нас є змога легко та ефективно пересуватися нашими структурами даних, і саме тут надійність структур даних Хаскела починає по-справжньому вражати.

14.5 Обережно — слизько

Дотепер, гуляючи нашими улюбленими структурами даних, чи то бінарними деревами, списками чи то файловими системами, ми безтурботно не боялися зайти занадто далеко і оступитися. До прикладу, наша функція `goLeft` брала застібку для бінарного дерева і пересувала фокус на ліве піддерево:

```
goLeft :: Zipper a -> Zipper a
goLeft (Node x l r, bs) = (l, LeftCrumb x r:bs)
```

А що, якби ми спробували спуститися пустим деревом? Іншими словами, як бути, якщо те дерево не `Node`, а

`Empty`? У цьому випадку ми отримаємо помилку виконання, бо зіставлення із взірцем не відбудеться, і в нас немає іншого взірця, що подбає про пусте дерево, у якого взагалі немає піддерев. Наразі ми просто припускали, що ми ніколи не сфокусуємося на лівому піддереві пустого дерева, бо в пустого дерева немає лівого піддерева. Просто наразі ми такий рух вважали безглуздим і зручнісінько ігнорували таку прикрість.

Ну а якщо б ми вже були у корені якогось дерева і навігаційна стежка була б порожня як банка, але ми все одно спробували б піти нагору? Відбулося б те саме. Здається, застібки річ така, що будь-який крок може стати нашим останнім (грає зловісна музика). Іншими словами, будь-який крок може бути успішним, але може бути і неуспішним також. Що це нам нагадує? Монади, звичайно ж! А конкретніше, це монада `Maybe`, що додає аварійного контексту до звичайних значень.

Ну то використаймо монаду `Maybe`, щоб додати аварійний контекст до наших рухів. Ми візьмемо наші функції, що працюють із застібками для бінарних дерев і перетворимо їх в монадні. Спершу, подбаймо про можливу аварію у `goLeft` і `goRight`. Дотепер, аварія функцій завжди віддзеркалювалася у їхньому результаті, і цього разу буде те саме. Ось `goLeft` і `goRight` із доданою можливістю аварійного завершення:

```
goLeft :: Zipper a -> Maybe (Zipper a)
goLeft (Node x l r, bs) = Just (l, LeftCrumb x r:bs)
goLeft (Empty, _) = Nothing

goRight :: Zipper a -> Maybe (Zipper a)
goRight (Node x l r, bs) = Just (r, RightCrumb x l:bs)
goRight (Empty, _) = Nothing
```

Класно, тепер якщо ми спробуємо піти ліворуч від пустого дерева, отримаємо `Nothing`!

```
ghci> goLeft (Empty, [])
Nothing
```

```
ghci> goLeft (Node 'A' Empty Empty, [])
Just (Empty,[LeftCrumb 'A' Empty])
```

Виглядає зовсім непогано. Як щодо руху нагору? Мали проблему із рухом нагору, коли хотіли піти наверх, а крихт не було, що означало, що ми були в корені дерева. Ось функція `goUp`, яка викидає помилку, якщо ми вийдемо за межі нашого дерева:

```
goUp :: Zipper a -> Zipper a
goUp (t, LeftCrumb x r:bs) = (Node x t r, bs)
goUp (t, RightCrumb x l:bs) = (Node x l t, bs)
```

Тепер модифікуймо її так, щоб вона завалювалась граціозно:

```
goUp :: Zipper a -> Maybe (Zipper a)
goUp (t, LeftCrumb x r:bs) = Just (Node x t r, bs)
goUp (t, RightCrumb x l:bs) = Just (Node x l t, bs)
goUp (_, []) = Nothing
```

Якщо є крихти, все окей і ми повертаємо успішний новий фокус, а як немає — повертаємо помилку.

Дотепер, ці функції брали застібки і повертали застібки, що дозволяло з'єднувати їх в ланцюг в отакий спосіб для пересування:

```
gchi> let newFocus = (freeTree,[]) -: goLeft -: goRight
```

Але тепер замість старого `Zipper a` вони повертають `Maybe (Zipper a)`, тому таке з'єднання в ланцюг не спрацює. У нас була схожа задача, коли ми возилися із нашим канатохідцем у підрозділі ?? з розділу про монади. Він також ходив один крок за раз і кожен із кроків міг бути фатальним, бо купа пташок могли приземлитися на одну сторону його жердяки для балансування і призвести до його падіння.

Тепер він мабуть сміється, бо тепер наша черга ходити, і ми блукаємо лабіринтами, які самі ж збудували. На жаль-на щастя, ми можемо повчитися у нашого канатохідця і робити те саме, що й він, а саме: замінити звичайне застосування функції на монадне `>>=`, що бере значення із контекстом (у нашому випадку, `Maybe (Zipper a)`, де контекст описує можливість аварії) і годує ним функцію, водночас дбаючи про правильне скористання тим контекстом, за потреби. Тому, за аналогією із канатохідцем, ми робимо бартерний обмін наших операторів `-:` на `>>=`. Добре, тепер таке з'єднання в ланцюг працюватиме знову! Дивіться:

```
ghci> let coolTree = Node 1 Empty (Node 3 Empty Empty)
ghci> return (coolTree,[]) >>= goRight
Just (Node 3 Empty Empty,[RightCrumb 1 Empty])
```

```
ghci> return (coolTree,[]) >>= goRight >>= goRight
Just (Empty,[RightCrumb 3 Empty,RightCrumb 1 Empty])
ghci> return (coolTree,[]) >>= goRight >>= goRight >>= goRight
Nothing
```

Ми використовуємо `return`, щоб загорнути застібку в `Just`, а після — `>>=`, щоб нагодувати нашу функцію `goRight`. Спочатку ми створили дерево з пустим лівим піддеревом, і непустим правим, що складається з вузла із двома пустими піддеревами. Коли ми спробували піти праворуч один раз, результат був успішним, бо операція мала сенс. Двічі праворуч працювало також; наш фокус вказував на пусте піддерево. А от іти праворуч тричі — то є нісенітниця, тому що ми не можемо піти праворуч, виходячи з пустого піддерева, і тому в результаті отримуємо `Nothing`.

Тепер ми підклали під нашими деревами монадні скирти сіна, і вони нас врятують якщо що. Круто.

Наша файлова система також має інші крайові випадки, де можуть бути аварії іншого типу, такі як, наприклад, спроба фокусування на файлі чи директорії, що не існують. Щоб трохи повправлятися, спробуйти озброїти нашу файлову систему функціями, що граціозно завалюються завдяки використанню монади `Maybe`.

Покажчик

Maybe
монада **Maybe**, 8, 18
breadcrumbs
навігаційна стежка; хлібні крихти, 11
breadcrumb
хлібна крихта, 11
break
функція **break**, 15
failure
аварія, 8
folder
директорія, 13
graceful failure
граціозна аварія, 19
immutable data structure
незмінення структура даних, 17
impure language; impure functional language
нечистофункційна мова, 1
internal node
вузол-серединка, 10
leaf node
вузол-листя, 10
leaf
листя, 10
monadic functions
монадні функції, 18
mutable data structure
зміненна структура даних, 17
persistence
персистентність, 17

sub-folder
піддиректорія, 13
sub-list
підсписок, 11
sub-tree
піддерево, 2
success
успішне завершення, 8
to chain
з'єднати в ланцюг, 19
type synonym
синонімічний тип; тип-синонім, 7
zipper
застібка; застібка-блискавка, 9
аварія
failure, 8
вузол-листя
leaf node, 10
вузол-серединка
internal node, 10
граціозна аварія
graceful failure, 19
директорія
folder, 13
з'єднати в ланцюг
to chain, 19
застібка; застібка-блискавка
zipper, 9
змінення структура даних
mutable data structure, 17
листя
leaf, 10
монада *Maybe*, 8, 18
монадні функції
monadic functions, 18
навігаційна стежка; хлібні крихти
breadcrumbs, 11
незмінення структура даних
immutable data structure, 17
нечистофункційна мова

impure language; impure functional language, 1
персистентність
persistence, 17
піддерево
sub-tree, 2
піддиректорія
sub-folder, 13
підсписок
sub-list, 11
синонімічний тип; тип-синонім
type synonym, 7
успішне завершення
success, 8
функція *break*, 15
хлібна крихта
breadcrumb, 11